Department of Computer Engineering

Academic Term: First Term 2

Class: T.E /Computer Sem – V / Software Engineering

Practical No:	6
Title:	Estimating Project Cost Using COCOMO Model in Software Engineering
Date of Performance:	5-09-23
Roll No:	9542
Team Members:	Chris Gracias

Rubrics for Evaluation:

Sr. No	Performance Indicator	Excellent	Good	Below Average	Total Score
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Theory Understanding (02)	02(Correct)	NA	01 (Tried)	

3	Content Quality (03)	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Questions (04)	04(done well)	3 (Partially Correct)	2(submitted)	

Signature of the Teacher:

Department of Computer Engineering
Academic Term: First Term 2022-23

Class: T.E /Computer Sem – V / Software Engineering

Lab Experiment 06

Experiment Name: Estimating Project Cost Using COCOMO Model in Software Engineering

Objective: The objective of this lab experiment is to introduce students to the COCOMO (Constructive Cost Model) estimation technique for estimating software project cost and effort. Students will gain practical experience in using the COCOMO model to estimate the development effort, duration, and resources required for a sample software project.

Introduction: COCOMO is a widely used algorithmic cost estimation model in software engineering. It helps in quantifying the effort and resources needed for software development based on project size, complexity, and other factors.

Lab Experiment Overview:

1. Introduction to COCOMO Model: The lab session begins with an introduction to the COCOMO model, explaining the different versions (Basic, Intermediate, and Advanced) and their application in software cost estimation.
2. Defining the Sample Project: Students are provided with a sample software project along with its functional and non-functional requirements, complexity, and size metrics.
3. COCOMO Parameters: Students learn about the COCOMO model parameters, such as Effort Adjustment Factor (EAF), Scale Factors, and Cost Drivers, and how they influence the project's effort estimation.

4. Effort and Duration Estimation: Using the COCOMO model formula, students estimate the effort and duration required to complete the sample project based on the provided size and complexity metrics.

5. Resource Allocation: Students estimate the number of required resources, such as developers, testers, and project managers, based on the calculated effort and project duration.

6. Sensitivity Analysis: Students perform sensitivity analysis by varying the COCOMO parameters to observe their impact on the project cost estimation.

7. Conclusion and Reflection: Students discuss the significance of COCOMO in software project estimation and reflect on their experience in estimating project cost using the COCOMO model.

Learning Outcomes: By the end of this lab experiment, students are expected to:

- Understand the COCOMO model and its application in software cost estimation.
- Gain practical experience in using the COCOMO model to estimate effort, duration, and resources for a software project.
- Learn to consider various project factors and adjust COCOMO parameters for accurate cost estimation.
- Develop estimation skills for resource allocation and project planning.
- Appreciate the importance of data accuracy and project size metrics in project cost estimation.

Pre-Lab Preparations: Before the lab session, students should familiarize themselves with the COCOMO model, its parameters, and the cost estimation formula. They should also review the factors that influence the project's size and complexity.

Materials and Resources:

- Project brief and details for the sample software project
- COCOMO model guidelines and cost estimation formula
- Calculators or spreadsheet software for performing calculations

Conclusion: The lab experiment on estimating project cost using the COCOMO model provide sstudents with practical insights into software cost estimation techniques. By applying the COCOMO model to a sample software project, students gain hands-on experience in assessing effort, duration, and resource requirements. The sensitivity analysis allows them to understand the impact of various factors on cost estimation. The lab experiment encourages students to use COCOMO in real-world scenarios, promoting informed decision-making in software project planning and resource allocation. Accurate cost estimation using COCOMO enhances project management and contributes to the successful execution of software engineering projects.

COCOMO

Basic model

1.organic:

KLOC=30

a=2.4 b=1.05

c=2.5 d=0.38

Formula:

$E=a(KLOC)^b$

$=2.4(30)^{1.05}$

effort=85 person-month

$T_{def}=c(E)^d$

$=2.5(85)^{0.38}$

Developing time=13 months

2.semi-detached:

a=3.0 b=1.12

c=2.5 d=0.35

Formula:

$E=a(KLOC)^b$

$=3.0(30)^{1.12}$

effort=135 person-month

$T_{def}=c(E)^d$

$=2.5(135)^{0.35}$

Developing time=14 months

3.Embedded:

a=3.6 b=1.20

c=2.5 d=0.32

Formula:

$E=a(KLOC)^b$

$=3.6(30)^{1.20}$

effort=213 person-month

$T_{def}=c(E)^d$

$=2.5(213)^{0.32}$

Developing time=14 months
Intermediate model:

1.ORGANIC:
KLOC=30
a=3.2 b=1.05
c=2.5 d=0.38
EAF=11.37
Formula:
$E=a(KLOC)^{b}*EAF$
$=3.2(30)^{1.05}*11.37$
effort=5978 person-month
$T_{def}=c(E)^{d}$
$=2.5(85)^{0.38}$
Developing time=68 months

2.SEMI-DETACHED:
KLOC=30 a=3.0
b=1.12 c=2.5 d=0.35
EAF=11.37 Formula:
$E=a(KLOC)^{b}*EAF$
$=3.0(30)^{1.12}*11.37$
effort=1539 person-month
$T_{def}=c(E)^{d}$
$=2.5(85)^{0.35}$
Developing time=32 months

3.EMBEDDED:
KLOC=30
a=2.8 b=1.20
c=2.5 d=0.32
EAF=11.37
Formula:
$E=a(KLOC)^{b}*EAF$
$=2.8(30)^{1.20}*11.37$
effort=1885 person-month
$T_{def}=c(E)^{d}$
$=2.5(85)^{0.32}$ Developing time=27 months

Postlabs:

Analyze a given software design and assess the level of cohesion and coupling, identifying potential areas for improvement.

Ans: I can provide a high-level analysis of cohesion and coupling in a software design, but I would need specific details about the software design to provide a more precise evaluation. Cohesion and coupling are important principles in software design:

1. **Cohesion**:

Cohesion refers to the degree to which the responsibilities of a module or component are related and focused. High cohesion implies that a module or component has a well-defined and single-purpose. Low cohesion suggests that a module or component is handling multiple unrelated responsibilities.

Areas of potential improvement regarding cohesion might include:

- **Functional Cohesion**: If a module is performing multiple functions that are not directly related, consider breaking it down into smaller, more focused modules.

- **Sequential Cohesion**: If a module's functions are only loosely related and depend on each other in a strict sequence, consider redesigning to reduce sequential coupling.

2. **Coupling**:

Coupling measures the level of interdependence between modules or components. Low coupling means that modules are more independent and can be modified without affecting many other parts of the system. High coupling suggests that changes in one module will necessitate changes in many others.

Areas of potential improvement regarding coupling might include:

- **Reduce Tight Coupling**: If modules are tightly coupled, refactor to use interfaces or abstract classes to create a more loosely coupled architecture.

- **Minimize Data Coupling**: When modules share data structures or have direct dependencies on each other's data, consider encapsulating data and providing access through well-defined interfaces.

- **Reduce Control Coupling**: If one module dictates the flow or control of another module, try to refactor to make the dependent module more self-contained and independent.

To provide a more detailed assessment and specific improvement recommendations, it would be helpful to have access to the software design or, at the very least, a description of the architecture and relationships between modules. This would allow for a more thorough analysis and actionable suggestions to improve cohesion and coupling within the software design.

Apply Object-Oriented principles, such as encapsulation and inheritance, to design a class hierarchy for a specific problem domain.

Ans: Certainly! Let's design a simple class hierarchy using object-oriented principles like encapsulation and inheritance for a problem domain. For this example, we'll create a class hierarchy to represent different types of vehicles.

1. **Base Class: Vehicle**

The base class 'Vehicle' will contain common properties and methods shared by all types of vehicles. ```python class Vehicle:

```
def __init__(self, make, model, year, price):
    self.make = make
self.model = model
```

self.year = year

self.price = price def

start(self):

    # Common method to start the vehicle print(f"The

{self.year} {self.make} {self.model} is starting.") def stop(self):

    # Common method to stop the vehicle print(f"The

{self.year} {self.make} {self.model} is stopping.")

```

2. **Inheritance: Car and Motorcycle**

We'll create two subclasses, 'Car' and 'Motorcycle', which inherit from the 'Vehicle' base class. Each subclass will have its own properties and methods in addition to the ones inherited from the 'Vehicle' class.

```python  class

Car(Vehicle):

  def __init__(self, make, model, year, price, num_doors):

    super().__init__(make, model, year, price)

self.num_doors = num_doors

def drive(self):

    print(f"The {self.year} {self.make} {self.model} with {self.num_doors} doors is driving.")

class Motorcycle(Vehicle):

  def __init__(self, make, model, year, price, style):

    super().__init__(make, model, year, price)

self.style = style

def ride(self):

    print(f"The {self.year} {self.make} {self.model} {self.style} is riding.")

```
```

Now, we have a class hierarchy for vehicles. The 'Vehicle' class serves as the base class, encapsulating common attributes and methods, and the 'Car' and 'Motorcycle' classes inherit from 'Vehicle' while adding their specific attributes and methods.

Here's how you can use this class hierarchy:

```python

# Create instances of Car and Motorcycle my_car = Car("Toyota", "Camry", 2022,

25000, 4) my_motorcycle = Motorcycle("Harley-Davidson", "Sportster", 2022, 12000,

"Cruiser")

# Call methods on the instances

my_car.start() my_car.drive()

my_motorcycle.start()

my_motorcycle.ride()
```

This design follows the principles of encapsulation by bundling data (attributes) and behaviors (methods) within classes and inheritance to model an "is-a" relationship between the base class 'Vehicle' and its subclasses 'Car' and 'Motorcycle'.

Evaluate the impact of cohesion and coupling on software maintenance, extensibility, and reusability in a real-world project scenario.

Ans: Cohesion and coupling are fundamental concepts in software design, and they have a significant impact on software maintenance, extensibility, and reusability in real-world project scenarios. Let's evaluate this impact:

1. **Impact on Software Maintenance**:

- **Cohesion**: High cohesion results in well-structured modules that are focused on a single, welldefined purpose. In real-world projects, this means that when you need to maintain or modify a specific feature or component, you can do so with minimal impact on other parts of the system. This makes maintenance more straightforward and less error-prone.

- **Coupling**: Low coupling between modules implies that they are less dependent on each other. In practical terms, this means that when you make changes to one module, it's less likely to introduce issues in other parts of the system. This leads to more predictable maintenance efforts.

2. **Impact on Software Extensibility**:

- **Cohesion**: High cohesion supports extensibility. When modules have a single responsibility, adding new features or functionality is easier. You can extend the system by adding new, cohesive modules without affecting existing ones.

- **Coupling**: Low coupling also enhances extensibility. A loosely coupled system allows you to plug in new components or modules without having to rewrite or modify existing code. This makes it easier to scale the software by adding new features or integrating with third-party libraries.

3. **Impact on Software Reusability**:

- **Cohesion**: High cohesion often leads to highly reusable components. When modules are designed with a single purpose in mind, they can be easily reused in other projects or parts of the same project. Reusable components save development time and effort.

- **Coupling**: Low coupling is crucial for reusability. If a module is loosely coupled with the rest of the system, it can be extracted and reused in various contexts without dragging along unnecessary dependencies. This promotes code reusability and the creation of libraries and frameworks.

Real-World Scenario:

Consider a real-world scenario in which you have a complex e-commerce platform. This platform consists of numerous components, such as the shopping cart, payment processing, user authentication, and product catalog. Each of these components should exhibit both high cohesion and low coupling:

- **Maintenance**: If the user authentication component has high cohesion, you can update it to support new authentication methods without affecting other parts of the system. Low coupling ensures that changes to authentication don't disrupt the shopping cart or payment processing.

- **Extensibility**: When you want to add a new payment gateway, high cohesion in the payment processing component allows you to do so without rewriting large portions of the code. Low coupling ensures that adding this new feature doesn't introduce bugs elsewhere.

- **Reusability**: High cohesion and low coupling in the product catalog component make it easily reusable in a different project or another section of the same e-commerce platform.